Hibernate and JPA

About me

Trayan Iliev

– CEO of **IPT – Intellectual Products & Technologies**
http://www.iproduct.org

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

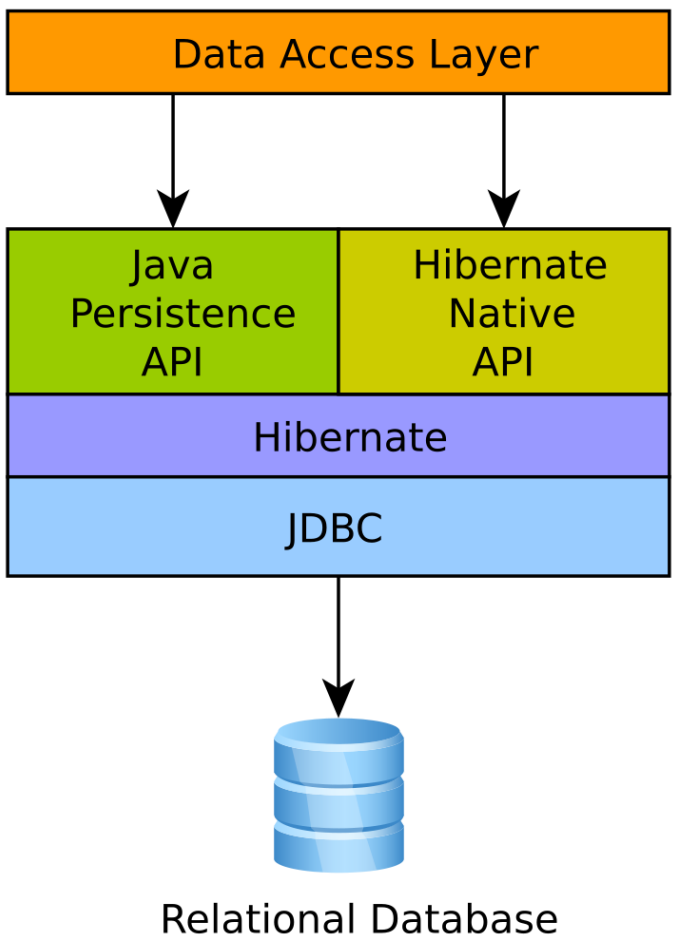
– Organizer RoboLearn hackathons and IoT enthusiast

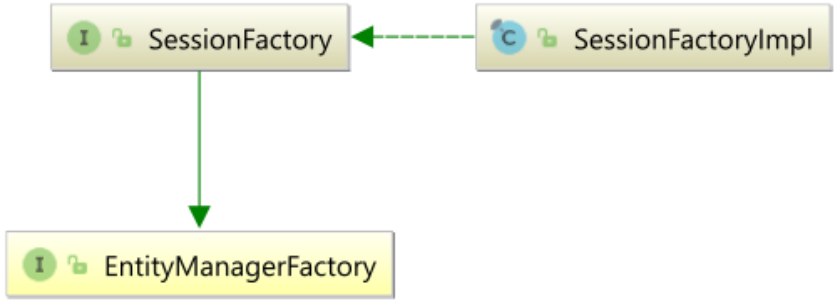
Where to Find The Code and Materials?

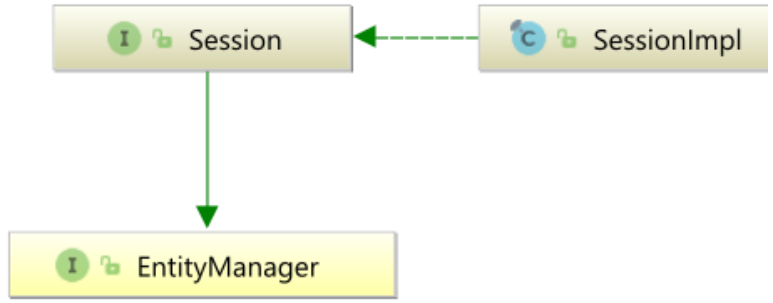
Java Academy projects and examples are available @GitHub:

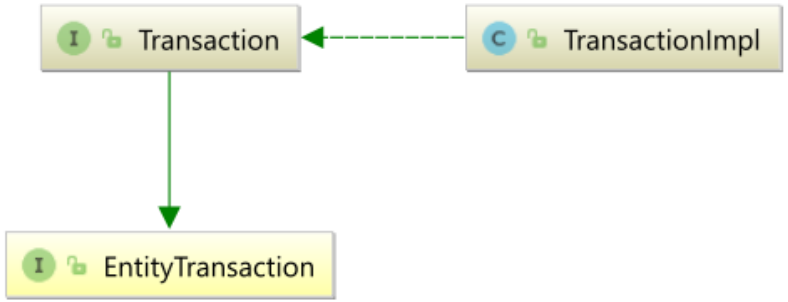
https://github.com/iproduct/java-spring-academy-2022.git

Hibernate Architecture









Hibernate Architecture - II

- **SessionFactory (org.hibernate.SessionFactory)** - a thread-safe (and immutable) representation of the mapping of the application domain model to a database. Acts as a factory for org.hibernate.Session instances. The EntityManagerFactory is the JPA equivalent of a SessionFactory and basically, those two converge into the same SessionFactory implementation.
- Very expensive to create, so, for any given database, the application should have only one associated SessionFactory.
- The SessionFactory maintains services that Hibernate uses across all Session(s) such as second level caches, connection pools, transaction system integrations, etc.

Hibernate Architecture - III

- **Session (org.hibernate.Session)** - a single-threaded, short-lived object conceptually modeling a "Unit of Work" (PoEAA). In JPA nomenclature, the Session is represented by an EntityManager.
- Behind the scenes, the Hibernate Session wraps a JDBC java.sql.Connection and acts as a factory for org.hibernate.Transaction instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.
- **Transaction (org.hibernate.Transaction)** - a single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries. EntityTransaction is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).

Domain Model

- Historically applications using Hibernate would have used its proprietary XML mapping file format for this purpose. With the coming of JPA, most of this information is now defined in a way that is portable across ORM/JPA providers using annotations (and/or standardized XML format).
- We usually prefer the JPA mappings where possible.
- For Hibernate mapping features not supported by JPA we will prefer Hibernate extension annotations.

Mapping Types

- Hibernate understands both the Java and JDBC representations of application data.
- Hibernate type – provides the ability to read/write this data from/to the database. It is an implementation of the org.hibernate.type.Type interface. Also describes various behavioral aspects of the Java type such as how to check for equality, how to clone values, etc.
- Hibernate type is **neither a Java type nor a SQL data type**. It provides information about mapping a Java type to an SQL type as well as how to persist and fetch a given Java type to and from a relational database.
- When you encounter the term type in discussions of Hibernate, it may refer to the Java type, the JDBC type, or the Hibernate type, depending on the context.

Hiberbate Native Bootstrapping

- There are two types of ServiceRegistry and they are hierarchical:
- **BootstrapServiceRegistry**, which has no parent and holds these three required services:
  - ClassLoaderService: allows Hibernate to interact with the ClassLoader of the various runtime environments
  - IntegratorService: controls the discovery and management of the Integrator service allowing third-party applications to integrate with Hibernate
  - StrategySelector: resolves implementations of various strategy contracts
- **StandardServiceRegistry**

Building BootstrapServiceRegistry

```
BootstrapServiceRegistry bootstrapServiceRegistry =
        new BootstrapServiceRegistryBuilder()
                .applyClassLoader()
                .applyIntegrator()
                .applyStrategySelector()
                .build();
```

Building StandardServiceRegistry

```
BootstrapServiceRegistry bootstrapServiceRegistry =
        new BootstrapServiceRegistryBuilder().build();
StandardServiceRegistryBuilder standardServiceRegistryBuilder =
        new StandardServiceRegistryBuilder(bootstrapServiceRegistry);
StandardServiceRegistry standardServiceRegistry = standardServiceRegistryBuilder
        .configure()
        .build();
```

Building Metadata

```
MetadataSources metadataSources =
        new MetadataSources(standardServiceRegistry);
metadataSources.addPackage( … );
metadataSources.addAnnotatedClass( … );
metadataSources.addResource( … )
Metadata metadata = metadataSources.buildMetadata();
```

Building and Using SessionFactory and Session

```
// Get SessionFactory
SessionFactory sessionFactory = metadata.getSessionFactoryBuilder().build();

// Get Session
Session session = sessionFactory.openSession();

// Persist entity
Contact contact = new Contact(1,
        new Name("Ivan", "Dimitrov", "Petrov"),
        "From work", new URL("http://ivan.petrov.me/"), true);
session.beginTransaction();
session.persist(contact);
session.getTransaction().commit();
session.close();
sessionFactory.close();
```

WEB-INF/applicationContext.xml -I

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd">

    <context:property-placeholder location="classpath:jdbc.properties" />

    <context:component-scan base-package="org.iproduct.spring.webmvc.dao,
         org.iproduct.spring.webmvc.service"/>

    <context:annotation-config />
```

Source: https://docs.spring.io/spring-framework/docs/

WEB-INF/applicationContext.xml -II

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
        destroy-method="close">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="mappingResources">
        <list><value>article.hbm.xml</value></list>
    </property>
    <property name="hibernateProperties">
        <value>
            hibernate.dialect=org.hibernate.dialect.HSQLDialect
            hibernate.hbm2ddl.auto=update
        </value>
    </property>
</bean>
```

Source: https://docs.spring.io/spring-framework/docs/

WEB-INF/applicationContext.xml III

```
   <bean id="transactionManager"
      class="org.springframework.orm.hibernate5.HibernateTransactionManager">
      <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <tx:annotation-driven/>

</beans>
```

Source: https://docs.spring.io/spring-framework/docs/

Hibernate Mapping: article.hbm.xml

```
<hibernate-mapping>
    <class name="org.iproduct.spring.webmvc.model.Article" table="ARTICLES">

        <meta attribute="class-description">
            This class contains the articles details.
        </meta>

        <id name="id" type="long" column="id">
            <generator class="identity"/>
        </id>

        <property name="title" column="title" type="string"/>
        <property name="content" column="content" type="string"/>
        <property name="createdDate" column="created_date" type="timestamp"/>
        <property name="pictureUrl" column="picture_url" type="string"/>

    </class>
</hibernate-mapping>
```

Source: https://docs.spring.io/spring-framework/docs/

Java Persistence API (JPA)

- **JPA four main parts:**
  - Java Persistence API
  - JPA Query Language
  - Java Persistence Criteria API
  - Object to Relational Mapping (ORM) metadata
- **JPA Entity Classes**
  - persistent fields
  - persistent properties
- **@Entity** annotation

Advantages of Spring ORM

- Easier testing
- Common data access exceptions
- General resource management
- Integrated transaction management

Persistent Units

- Persistent Unit description in persistence.xml file:

–description
–provider
–jta-data-source
–non-jta-data-source
–mapping-file
–jar-file
–class
–exclude-unlisted-classes
–properties

Persistent Unit Example 1

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
    <persistence-unit name="CustomerDBPU" transaction-type="JTA">
        <jta-data-source>jdbc/sample</jta-data-source>
        <class>customerdb.Customer</class>
        <class>customerdb.DiscountCode</class>
        <properties/>
    </persistence-unit>
</persistence>
```

Persistent Unit Example 2

```
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
   <persistence-unit name="invoicingPU" transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.PersistenceProvider</provider>
      <class>myinvoice.dbentities.Product</class>
      <class>myinvoice.dbentities.Invoice</class>
      <properties>
         <property name="javax.persistence.jdbc.user" value="root"/>
         <property name="javax.persistence.jdbc.password" value="root"/>
         <property name="javax.persistence.jdbc.url“
                        value="jdbc:mysql://localhost:3306/invoicing"/>
         <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      </properties>
   </persistence-unit>
</persistence>
```

JPA Setup in Spring

```
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="myPersistenceUnit"/>
    </bean>
</beans>
```

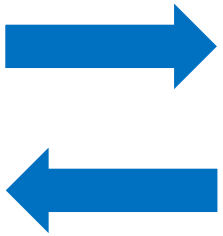
```
<beans>
    <jee:jndi-lookup id="myEmf" jndi-name="persistence/myPersistenceUnit"/>
</beans>
```

```
<beans>
    <bean id="myEmf" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="someDataSource"/>
        <property name="loadTimeWeaver">
            <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
        </property>
    </bean>
</beans>
```

Mapping Example

```
@Entity(name = "Contact")
@Data
public class Contact {

    @Id
    private Integer id;

    @Embedded
    private Name name;

    private String notes;

    private URL website;

    private boolean starred;

    public Name getName() {
        return name;
    }
}
```

```
@Embeddable
@Data
public class Name {

    private String firstName;

    private String middleName;

    private String lastName;
}
```



```
create table Contact
(
    id      integer not null,
    first   varchar(255),
    last    varchar(255),
    middle  varchar(255),
    notes   varchar(255),
    starred boolean not null,
    website varchar(255),
    primary key (id)
)
```

Code First

DB Schema First

Value and Entity Types

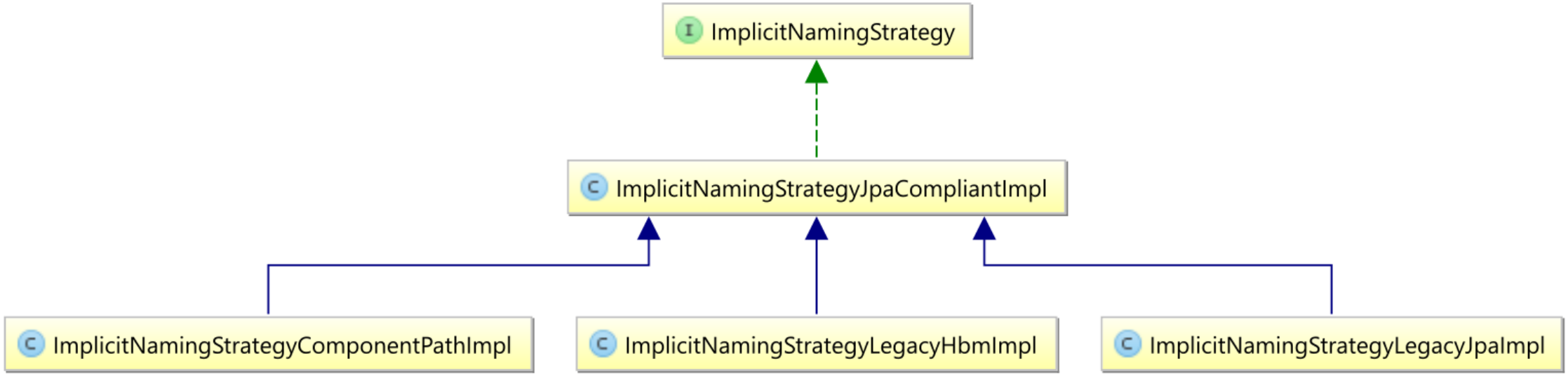
- **Value types** - a value type is a piece of data that does not define its own lifecycle. It is, in effect, owned by an entity, which defines its lifecycle -> persistent attributes.
  - Basic types – e.g. in mapping the Contact table, all attributes except for name would be basic types.
  - Embeddable types - the name attribute is an example of an embeddable type, which is discussed in details in Embeddable types
  - Collection types - although not featured in the aforementioned example, collection types are also a distinct category among value types. Collection types are
- **Entity types** - by nature of their unique identifier, entities exist independently and define their own lifecycle, whereas values do not. Entities are domain model classes which correlate to rows in a database table, using a unique identifier.

Naming Strategies

Part of the mapping of an object model to the relational database is mapping names from the object model to the corresponding database names. Hibernate looks at this as 2-stage process:

1. The first stage is determining a proper logical name from the domain model mapping. A logical name can be either explicitly specified by the user (e.g., using @Column or @Table) or it can be implicitly determined by Hibernate through an ImplicitNamingStrategy contract.
2. Second is the resolving of this logical name to a physical name which is defined by the PhysicalNamingStrategy contract.

Naming Strategies



Implicite Naming Strategies

- Hibernate defines multiple ImplicitNamingStrategy implementations out-of-the-box. Applications are also free to plug in custom implementations.
- Applications can specify the implementation using the hibernate.implicit_naming_strategy configuration setting which accepts:
  - **default** – for org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl - an alias for jpa
  - **jpa** - for org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl - the JPA 2.0 compliant naming strategy
  - **legacy-hbm** - for org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyHbmImpl - compliant with the original Hibernate NamingStrategy
  - **legacy-jpa** - for org.hibernate.boot.model.naming.ImplicitNamingStrategyLegacyJpaImpl - compliant with the legacy NamingStrategy developed for JPA 1.0, which was unfortunately unclear in many respects regarding implicit naming rules
  - **component-path** –org.hibernate.boot.model.naming.ImplicitNamingStrategyComponentPathImpl - mostly follows ImplicitNamingStrategyJpaCompliantImpl rules, except that it uses the full composite paths, as opposed to just the ending property part – e.g.
- By calling org.hibernate.boot.MetadataBuilder#applyImplicitNamingStrategy

Physical Naming Strategies

- There are multiple ways to specify the PhysicalNamingStrategy to use. First, applications can specify the implementation using the hibernate.physical_naming_strategy configuration setting which accepts:
  - reference to a Class that implements the org.hibernate.boot.model.naming.PhysicalNamingStrategy contract
  - FQN of a class that implements the org.hibernate.boot.model.naming.PhysicalNamingStrategy contract
- Secondly, applications and integrations can leverage org.hibernate.boot.MetadataBuilder#applyPhysicalNamingStrategy

Basic Types

- Basic value types usually map a single database column, to a single, non-aggregated Java type
- Internally Hibernate uses a registry of basic types when it needs to resolve a specific org.hibernate.type.Type:

https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html#basic-provided

- The @Basic annotation - defines 2 attributes:
  - optional - boolean (defaults to true) - whether this attribute allows nulls.
  - fetch - FetchType (defaults to EAGER) - whether should be fetched eagerly or lazily. Hibernate ignores this setting for basic types unless you are using bytecode enhancement.

Explicite Basic Types

```
@Entity(name = "Product")
 public class Product {
        @Id
        private Integer id; private String sku;

        @org.hibernate.annotations.Type( type = "nstring" )
        private String name;

        @org.hibernate.annotations.Type( type = "materialized_nclob" )
        private String description;
}
```

Basic Types

- Basic value types usually map a single database column, to a single, non-aggregated Java type
- Internally Hibernate uses a registry of basic types when it needs to resolve a specific org.hibernate.type.Type:

https://docs.jboss.org/hibernate/orm/5.6/userguide/html_single/Hibernate_User_Guide.html#basic-provided

- The @Basic annotation - defines 2 attributes:
  - optional - boolean (defaults to true) - whether this attribute allows nulls.
  - fetch - FetchType (defaults to EAGER) - whether should be fetched eagerly or lazily. Hibernate ignores this setting for basic types unless you are using bytecode enhancement.

Embeddable Types

```java
@Embeddable
public class Publisher {

    private String name;


    @Embedded
    private Location location;

    public Publisher(String name, Location location) {
        this.name = name;
        this.location = location;
    }

    private Publisher() {}

    //Getters and setters are omitted for brevity
}
```

```java
@Embeddable
public class Location {

    private String country;

    private String city;

    public Location(String country, String city) {
        this.country = country;
        this.city = city;
    }

    private Location() {}

    //Getters and setters are omitted for brevity
}
```

Embeddable Types - II

```java
@Entity(name = "Book")
public class Book {

    @Id
    @GeneratedValue
    private Long id;

    private String title;

    private String author;

    @Embedded
    private Publisher2 publisher;

    public Publisher2 getPublisher() {
        return publisher;
    }
}
```

```java
@Embeddable
@Data
class Publisher2 {

    @Column(name = "publisher_name")
    private String name;

    @Column(name = "publisher_country")
    private String country;
}
```

Embeddable Types – Attribute Overrides

```
@Entity(name = "Book")
@AttributeOverrides({
        @AttributeOverride(
                name = "ebookPublisher.name",
                column = @Column(name = "ebook_publisher_name")
        ),
        @AttributeOverride(
                name = "paperBackPublisher.name",
                column = @Column(name = "paper_back_publisher_name")
        )
})
@AssociationOverrides({
        @AssociationOverride(
                name = "ebookPublisher.country",
                joinColumns = @JoinColumn(name = "ebook_publisher_country_id")
        ),
        @AssociationOverride(
                name = "paperBackPublisher.country",
                joinColumns = @JoinColumn(name = "paper_back_publisher_country_id")
        )
})
```

```
public class Book2 {
    @Id
    @GeneratedValue
    private Long id;

    private String title;

    private String author;

    @Embedded
    private Publisher2 ebookPublisher;

    @Embedded
    private Publisher2 paperBackPublisher;

    public Publisher2 getPaperBackPublisher() {
        return paperBackPublisher;
    }

    public Publisher2 getEbookPublisher() {
        return ebookPublisher;
    }
}
```

@Target Mapping

```
@Embeddable
class GPS implements Coordinates {
    private double latitude;
    private double longitude;

    public GPS() {
    }

    public GPS(double latitude, double longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }
    @Override
    public double x() {
        return latitude;
    }
    @Override
    public double y() {
        return longitude;
    }
}
```

```
interface Coordinates {
    double x();
    double y();
}
```

```
@Entity(name = "City")
public class City {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @Embedded
    @Target( GPS.class )
    private Coordinates coordinates;
}
```

@Parent Mapping

```
@Embeddable
@Data
public class GPS {

    private double latitude;

    private double longitude;

    @Parent
    private City city;
}
```

```
entityManager -> {
    City cluj = new City(); cluj.setName( "Cluj" );
    cluj.setCoordinates( new GPS( 46.77120, 23.62360 ) );
    entityManager.persist( cluj );
    City cluj = entityManager.find( City.class, 1L );
    assertSame( cluj, cluj.getCoordinates().getCity() );
}
```

```
@Entity(name = "City")
@Data
public class City {

    @Id
    @GeneratedValue
    private Long id;

    private String name;

    @Embedded
    @Target( GPS.class )
    private GPS coordinates;
}
```

@Entity and @Table Mappings

```
@Entity(name = "Book")
@Table( catalog = "public", schema = "store", name = "book" )
public static class Book {
    @Id
    private Long id;
    private String title;
    private String author;
    …
}
```

```
create table public.book (
  id bigint not null,
  author varchar(255),
  title varchar(255),
  primary key (id)
) engine=InnoDB
```

Identifiers

- UNIQUE - the values must uniquely identify each row.
- NOT NULL - the values cannot be null. For composite ids, no part can be null.
- IMMUTABLE - the values, once inserted, can never be changed. This is more a general guide, than a hard-fast rule as opinions vary. JPA defines the behavior of changing the value of the identifier attribute to be undefined; Hibernate simply does not support that. In cases where the values for the PK you have chosen will be updated, Hibernate recommends mapping the mutable value as a natural id, and use a surrogate id for the PK.
- EVER-INCREASING - fragmentation problem - because UUIDs are random, they have no natural ordering so cannot be used for clustering. This is why SQL Server has implemented a newsequentialid() function that is suitable for use in clustered indexes, and UUID PKs.

Entity Objects Identity – equals() and hashCode()

```
Book book1 = new Book();
book1.setTitle("High-Performance Java Persistence");

Book book2 = new Book();
book2.setTitle("Java Persistence with Hibernate");

Library library = doInJPA(entityManager -> {
    Library _library = entityManager.find(Library.class, 1L);

    entityManager.persist(book1);
    entityManager.persist(book2);
    entityManager.flush();

    _library.getBooks().add(book1);
    _library.getBooks().add(book2);

    return _library;
});

assertTrue(library.getBooks().contains(book1));
assertTrue(library.getBooks().contains(book2));
```

Entity Objects Identity – equals() and hashCode() - II

```
@Entity(name = "Library")
public class Library {
    @Id
    private Long id;
    private String name;
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "book_id")
    private Set<Book> books = new HashSet<>();
}

@Entity(name = "Book")
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private String author;
    @NaturalId
    private String isbn;

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Book book = (Book) o;
        return Objects.equals( isbn, book.isbn );
    }

    @Override
    public int hashCode() {
        return Objects.hash( isbn );
    }
}
```

Entity Objects Identity – equals() and hashCode() - II

```
@Entity(name = "Library")
public class Library {
    @Id
    private Long id;
    private String name;
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "book_id")
    private Set<Book> books = new HashSet<>();
}

@Entity(name = "Book")
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    private String title;
    private String author;
    @NaturalId
    private String isbn;

    @Override
    public boolean equals(Object o) {
        if ( this == o ) {
            return true;
        }
        if ( o == null || getClass() != o.getClass() ) {
            return false;
        }
        Book book = (Book) o;
        return Objects.equals( isbn, book.isbn );
    }

    @Override
    public int hashCode() {
        return Objects.hash( isbn );
    }
}
```

Object-Relational Mapping (ORM)

- Package: javax.persistence
- Simple keys - @Id annotation
- Composite keys
  - Primary Key Class – requirements and structure
  - Annotations – @EmbeddedId, @IdClass
- Realtions between entity objects –
  - uni- and bi-directional,
  - 1:1, 1:many, many:1 many:many

Main JPA Annotations

- @PersistenceUnit,
- @PersistenceContext
- @Entity
- @Id
- @OneToOne
- @OneToMany
- @ManyToMany
- @DiscrimainatorColumn
- @Column
- @JoinTable
- @JoinColumn
- @Embeddable
- @Embedded

Entity Embeddables

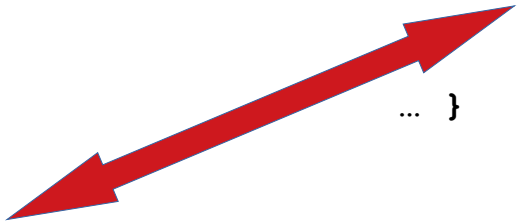
- **@Embeddable** – annotates class that is a value type (not Entity), but can be embedded into one or more Entities
- **@Embedded** – embeds Embeddable class into Entity class
- Embedding can be hierarchical on multiple levels
- Annotations: **@AttributeOverride, @AttributeOverrides, @AssociationOverride, @AssociationOverrides**

Collection Type Persistent Fields

- Field or properties should be of Collection or Map type (usually generic):
  - java.util.Collection
  - java.util.Set
  - java.util.List
  - java.util.Map
- @ElementCollection
- @CollectionTable – name of additional table
- @Embeddable, @Column
- @AttributeOverride, @AttributeOverrides

JPA Entities: @ManyToMany

```
@Entity
public class Book {
    @Id @GeneratedValue
    private int id;

    @NotNull
    private String title;

    @ManyToOne
    @JoinColumn(name = "PUBLISHER_ID",
                referencedColumnName = "id")
    private Publisher publisher;

    @Column(name = "PUBLISHED_DATE") @PastOrPresent
    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE)
    private LocalDate publishedDate;

    @Pattern(regexp = "\\d{10}|\\d{13}")
    private String isbn;

    @NotNull @Min(0)
    private double price;

    @ManyToMany(fetch = FetchType.EAGER)
    @JoinTable(name="BOOK_AUTHOR", joinColumns=
      @JoinColumn(name="BOOK_ID",referencedColumnName="ID"),
                inverseJoinColumns=
      @JoinColumn(name="AUTHOR_ID",referencedColumnName="ID")
    )
    private List<Author> authors = new ArrayList<>();
… }
```



```
@Entity
public class Author {
        @Id @GeneratedValue
        private int id;

        @NotNull
        @Length(min=2, max=60)
        @Column(name = "first_name")
        private String firstName;

        @NotNull
        @Length(min=2, max=60)
        @Column(name = "last_name")
        private String lastName;

        @ManyToMany(mappedBy = "authors",
                    fetch = FetchType.EAGER)
        List<Book> books = new ArrayList<>();
… }
```

Optimizers

- None - no optimization is performed. We communicate with the database each and every time an identifier value is needed from the generator.
- pooled-lo - The pooled-lo optimizer works on the principle that the increment-value is encoded into the database table/sequence structure. In sequence-terms, this means that the sequence is defined with a greater-than-1 increment size.
- pooled - just like pooled-lo, except that here the value from the table/ sequence is interpreted as the high end of the value pool.
- hilo; legacy-hilo - custom algorithm for generating pools of values based on a single value from a table or sequence.
- Applications can also implement and use their own optimizer strategies, as defined by the org.hibernate.id.enhanced.Optimizer contract.

ORM Cascade Updates

- Entities that have a dependency relationship can be managed declaratively by JPA using CascadeType:
  - **ALL** – всички операции са каскадни
  - **DETACH** – каскадно отстраняване
  - **MERGE** – каскадно сливане
  - **PERSIST** – каскадно персистиране
  - **REFRESH** – каскадно обновяване
  - **REMOVE** – каскадно премахване

```
@OneToMany(cascade=REMOVE, mappedBy="customer")
public Set<Order> getOrders() { return orders; }
```

ArticlesDaoHibernate Class - I

```
@Repository
@Transactional
public class ArticleDaoHibernate implements ArticleDao {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Override
    public Collection<Article> findAll() {
        return this.sessionFactory.getCurrentSession()
          .createQuery("select article from Article article",
Article.class)
          .list();
    }
    @Override
    public Article find(long id) {
        return this.sessionFactory.getCurrentSession()
          .byId(Article.class).load(id);
    }
```

Source: https://docs.spring.io/spring-framework/docs/

JPA Entity Annotations Example

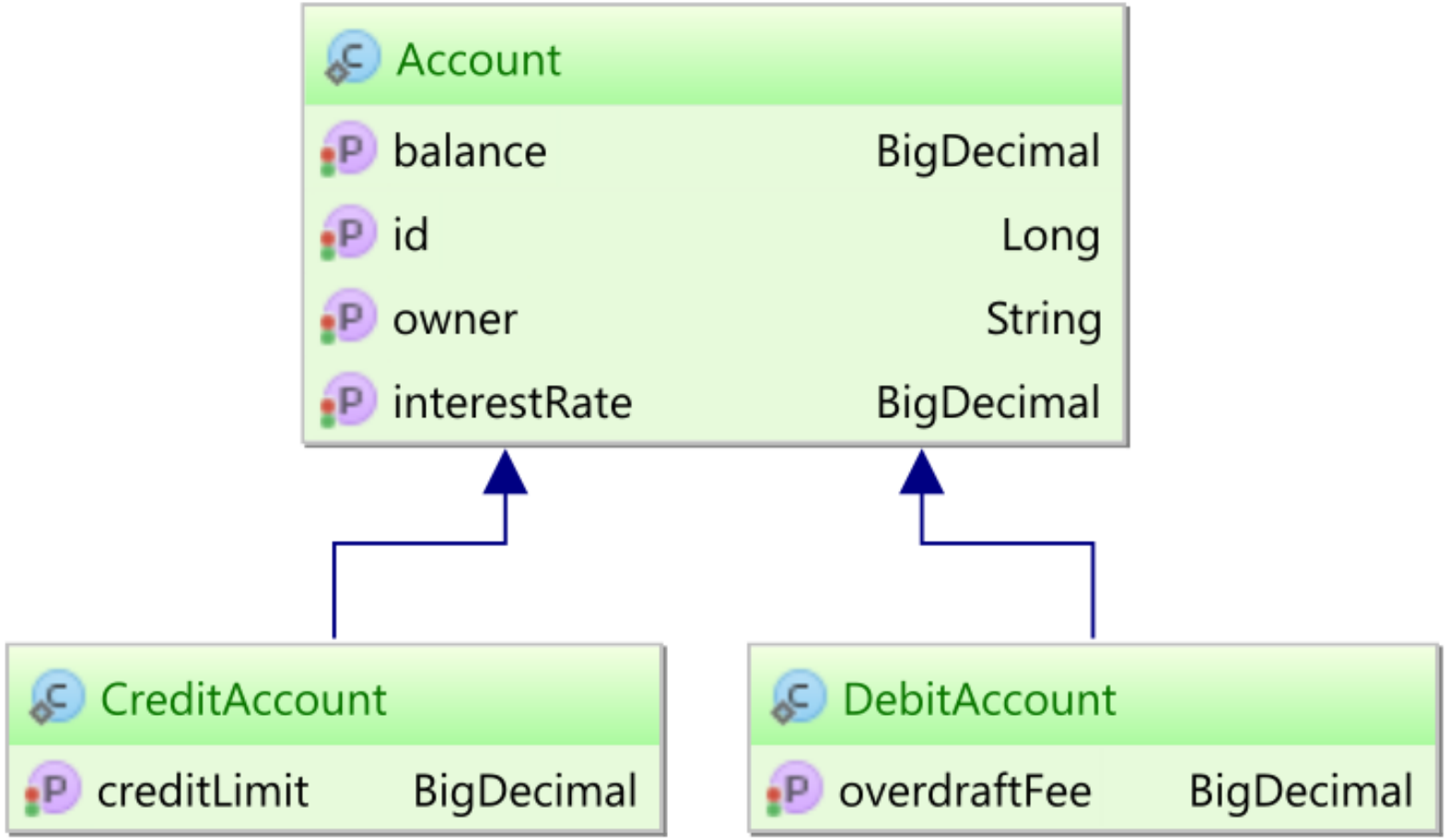
```
@Entity
public class Article {
    @Id
    @GeneratedValue
    private Long id;

    @Length(min=3, max=80)
    private String title;

    @Length(min=3, max=2048)
    private String content;

    @NotNull
    @ManyToOne
    @JoinColumn(name="AUTHOR_ID", nullable=false)
    private User author;

    @Length(min=3, max=256)
    private String pictureUrl;

    @Temporal(TemporalType.TIMESTAMP)
    private Date created = new Date();

    @Temporal(TemporalType.TIMESTAMP)
    private Date updated = new Date();
... }
```

```
@Entity
public class User implements UserDetails {
    @Id
    @GeneratedValue
    private long id;

    @NotNull
    @Length(min = 3, max = 30)
    private String username;
    ...

    @NonNull
    private String roles = "ROLE_USER";

    @OneToMany(mappedBy = "author",
               cascade = CascadeType.ALL,
               orphanRemoval=true)
    Collection<Article> articles =
        new ArrayList<>();

    @Temporal(TemporalType.TIMESTAMP)
    private Date created = new Date();

    @Temporal(TemporalType.TIMESTAMP)
    private Date updated = new Date();
... }
```

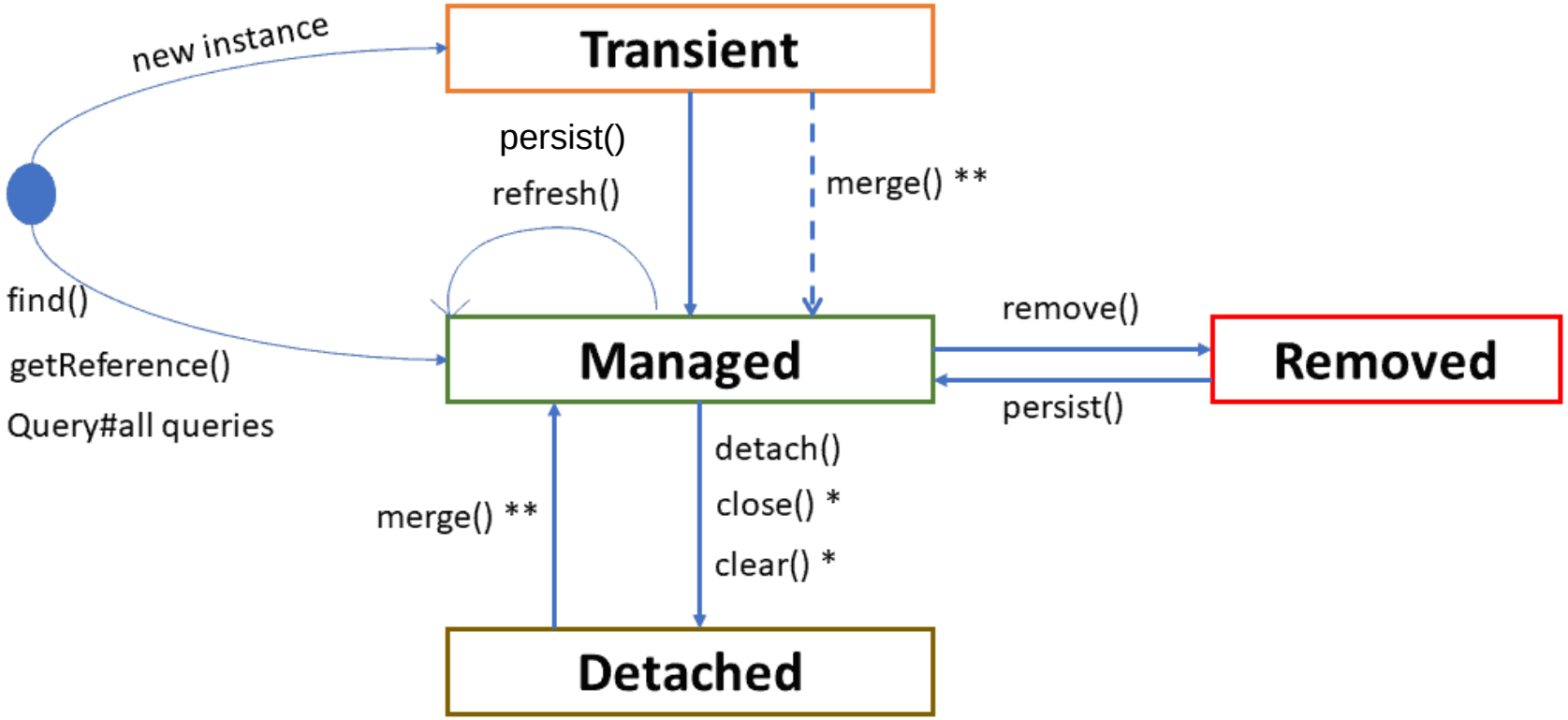
Entity Inheritance

- Entity / Abstract entity
- Mapped superclass
- Non-entity superclass
- Entity -> DB tables mapping strategies
  - –SingleTable per Class Hierarchy
  - –TheTable per Concrete Class
  - –The Joined Subclass Strategy

Entity Inheritance



JPA Entity Lifecycle

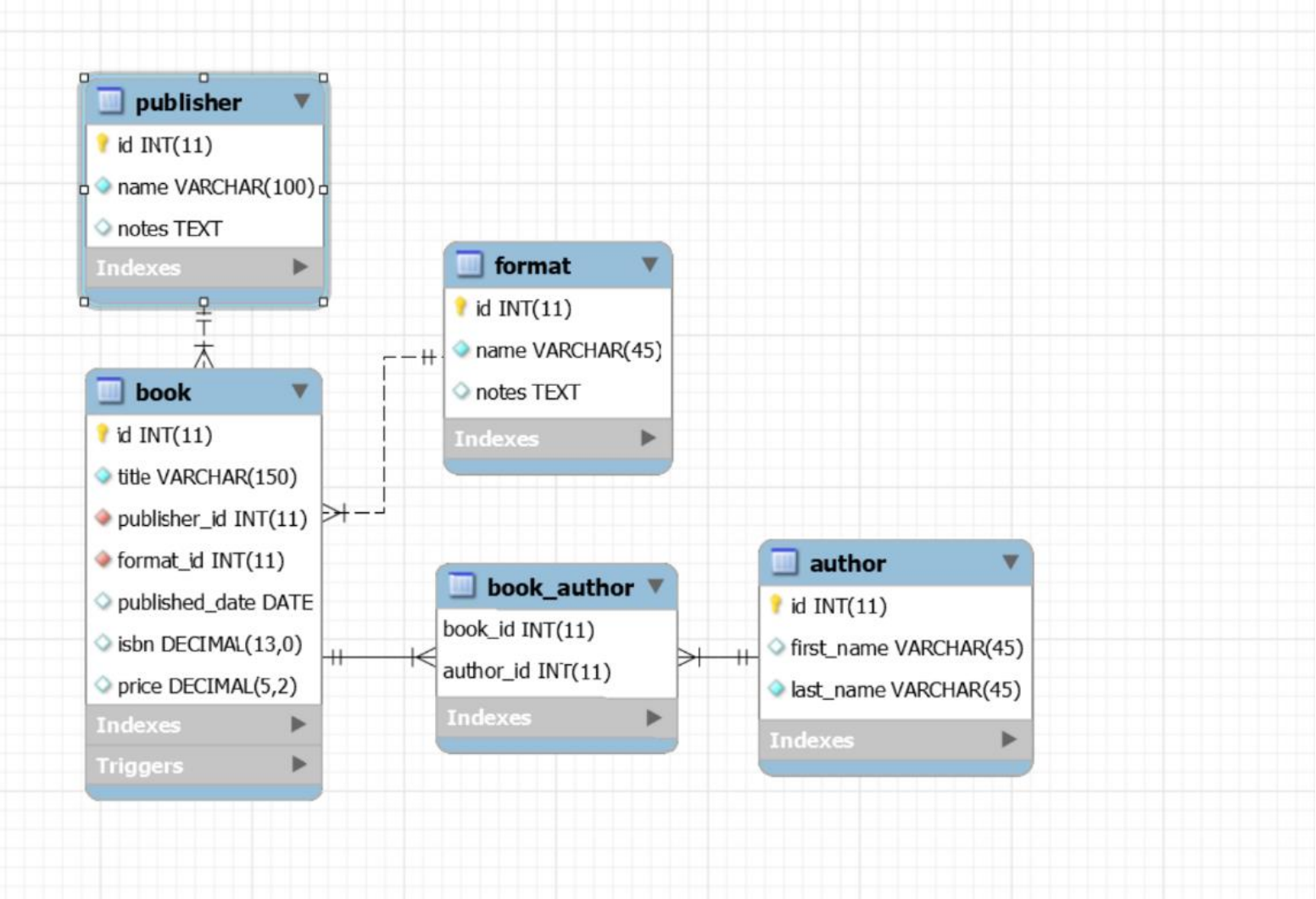


ArticlesDaoHibernate Class - II

```
@Override
public Article create(Article article) {
    this.sessionFactory.getCurrentSession()
            .persist(article);
    return article;
}
@Override
public Article update(Article article) {
    Article toBeDeleted = find(article.getId());
    if (toBeDeleted == null) {
        throw new EntityNotExistException("Article "+article.getId()+" not exist.");
    }
    return (Article) this.sessionFactory.getCurrentSession()
        .merge(article);
}
@Override
public Article remove(long articleId) {
    Article toBeDeleted = find(articleId);
    if (toBeDeleted == null) {
       throw new EntityNotExistException("Article "+article.getId()+" not exist.");
    }
    this.sessionFactory.getCurrentSession()
        .delete(toBeDeleted);
    return toBeDeleted;
}}
```

Source: https://docs.spring.io/spring-framework/docs/

JPA Entities: ER Diagram



JPA Query Language Syntax

- **Select Statements** - SELECT, FROM, WHERE, GROUP BY, HAVING, and ORDER BY.
- The SELECT clause defines the types of the objects or values returned by the query.
- The FROM clause defines the scope of the query by declaring one or more identification variables, which can be referenced in the SELECT and WHERE clauses. An identification variable represents one of the following elements:
  - The abstract schema name of an entity
  - An element of a collection relationship
  - An element of a single-valued relationship
  - A member of a collection that is the multiple side of a one-to-many relationship
- The WHERE clause is a conditional expression that restricts the objects or values retrieved by the query. Although the clause is optional, most queries have a WHERE clause.
- The GROUP BY clause groups query results according to a set of properties.
- The HAVING clause is used with the GROUP BY clause to further restrict the query results according to a conditional expression.
- The ORDER BY clause sorts the objects or values returned by the query into a specified order.

JPA Query Language Syntax

Update and delete statements provide bulk operations over sets of entities. These statements have the following syntax:

- update_statement :: = update_clause [where_clause]
- delete_statement :: = delete_clause [where_clause]
- The update and delete clauses determine the type of the entities to be updated or deleted. The WHERE clause may be used to restrict the scope of the update or delete operation.

Java Persistence Query Language

- Object-oriented database queries
- Navigation
- Abstract schema
- Path expression
- State field
- Relationship field

Java Persistence Query Language

- SELECT
- FROM
- WHERE
- GROUP BY
- HAVING
- ORDER BY
- UPDATE
- DELETE
- AS, IN
- LIKE
- EXISTS, ANY, ALL
- NEW

Basic JPA Query usage

```
Query query = entityManager.createQuery(
                    "select p " +
                        "from Person p " +
                        "where p.name like :name")
                // timeout - in milliseconds
                .setHint("javax.persistence.query.timeout", 2000)
                // flush only at commit time
                .setFlushMode(FlushModeType.COMMIT);
```

Hibernate Flush Modes

ALWAYS

Flushes the Session before every query.

AUTO

This is the default mode, and it flushes the Session only if necessary.

COMMIT

The Session tries to delay the flush until the current Transaction is committed, although it might flush prematurely too.

MANUAL

The Session flushing is delegated to the application, which must call Session.flush() explicitly in order to apply the persistence context changes.

JPA defines some standard hints - I

javax.persistence.query.timeout - Defines the query timeout, in milliseconds.

javax.persistence.fetchgraph - Defines a fetchgraph EntityGraph. Attributes explicitly specified as AttributeNodes are treated as FetchType.EAGER (via join fetch or subsequent select). For details, see the EntityGraph discussions in Fetching.

javax.persistence.loadgraph - Defines a loadgraph EntityGraph. Attributes explicitly specified as AttributeNodes are treated as FetchType.EAGER (via join fetch or subsequent select). Attributes that are not specified are treated as FetchType.LAZY or FetchType.EAGER depending on the attribute's definition in metadata. For details, see the EntityGraph discussions in Fetching.

org.hibernate.cacheMode - Defines the CacheMode to use. See org.hibernate.query.Query#setCacheMode.

org.hibernate.cacheable - Defines whether the query is cacheable. true/false. See org.hibernate.query.Query#setCacheable.

JPA defines some standard hints - II

org.hibernate.cacheRegion - For queries that are cacheable, defines a specific cache region to use. See org.hibernate.query.Query#setCacheRegion.

org.hibernate.comment - Defines the comment to apply to the generated SQL. See org.hibernate.query.Query#setComment.

org.hibernate.fetchSize - Defines the JDBC fetch-size to use. See org.hibernate.query.Query#setFetchSize.

org.hibernate.flushMode - Defines the Hibernate-specific FlushMode to use. See org.hibernate.query.Query#setFlushMode. If possible, prefer using javax.persistence.Query#setFlushMode instead.

org.hibernate.readOnly - Defines that entities and collections loaded by this query should be marked as read-only. See org.hibernate.query.Query#setReadOnly.

JPA retrieving result set

In terms of execution, JPA Query offers 3 different methods for retrieving a result set:

Query#getResultList() - executes the select query and returns back the list of results.

Query#getResultStream() - executes the select query and returns back a Stream over the results.

Query#getSingleResult() - executes the select query and returns a single result. If there were more than one result an exception is thrown.

Basic Hibernate Query usage

- org.hibernate.query.Query query = session.createQuery(

```
                "select p " +
                    "from Person p " +
                    "where p.name like :name")
            // timeout - in seconds
            .setTimeout(2)
            // write to L2 caches, but do not read from them
            .setCacheMode(CacheMode.REFRESH)
             // assuming query cache was enabled for the SessionFactory
            .setCacheable(true)
            // add a comment to the generated SQL if enabled via the
           // hibernate.use_sql_comments configuration property
            .setComment("+ INDEX(p idx_person_name)");
```

Hibernate query scrolling

```
try ( ScrollableResults scrollableResults = session.createQuery(
            "select p " +
                "from Person p " +
                "where p.name like :name" )
        .setParameter( "name", "J%" )
        .scroll()
) {
    while(scrollableResults.next()) {
        Person person = (Person) scrollableResults.get()[0];
        process(person);
    }
}
```

Hibernate query streaming

```
try ( Stream<Object[]> persons = session.createQuery(
                "select p.name, p.nickName " +
                        "from Person p " +
                        "where p.name like :name" )
        .setParameter( "name", "J%" )
        .stream() ) {

    persons
            .map( row -> new PersonNames(
                    (String) row[0],
                    (String) row[1] ) )
            .forEach( this::process );
}
```

Hibernate Fetching

Fetching

The concept of fetching breaks down into two different questions:

- When should the data be fetched? Now? Later?
- How should the data be fetched?

Fetching Options with Hibernate – Static:

Static definition of fetching strategies is done in the mappings. The statically-defined fetch strategies are used in the absence of any dynamically defined strategies.

- SELECT - performs a separate SQL select to load the data. This can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed). This trategy is generally termed N+1.
- JOIN - inherently an EAGER style of fetching. The data to be fetched is obtained through the use of an SQL outer join.
- BATCH - performs a separate SQL select to load a number of related data items using an IN-restriction as part of the SQL WHERE-clause based on a batch size. Again, this can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed).
- SUBSELECT - performs a separate SQL select to load associated data based on the SQL restriction used to load the owner. Again, this can either be EAGER (the second select is issued immediately) or LAZY (the second select is delayed until the data is needed).

Fetching Options with Hibernate – Dynamic:

Dynamic (sometimes referred to as runtime) definition is really use-case centric. There are multiple ways to define dynamic fetching:

- fetch profiles - defined in mappings, but can be enabled/disabled on the Session.
- HQL / JPQL - both Hibernate and JPA Criteria queries have the ability to specify fetching, specific to said query.
- entity graphs - starting in Hibernate 4.2 (JPA 2.1), this is also an option there are two types of entity graphs:
  - load graph – In this case, all attributes specified in the entity graph will be treated as FetchType.EAGER, but attributes not specified use their static mapping specification.
  - fetch graph - In this case, all attributes specified in the entity graph will be treated as FetchType.EAGER, and all attributes not specified will **ALWAYS** be treated as FetchType.LAZY.

Fetching Options with Hibernate

- The Hibernate recommendation is to statically mark all associations lazy and to use dynamic fetching strategies for eagerness.
- This is unfortunately at odds with the JPA specification which defines that all one-to-one and many-to-one associations should be eagerly fetched by default. Hibernate, as a JPA provider, honors that default.

No Fetching

```
Integer accessLevel = entityManager.createQuery(
            "select e.accessLevel " +
                "from Employee e " +
                "where " +
                "       e.username = :username and " +
                "       e.password = :password",
            Integer.class)
        .setParameter( "username", username)
        .setParameter( "password", password)
        .getSingleResult();
```

Dynamic fetching via queries

```
Employee employee = entityManager.createQuery(
            "select e " +
                  "from Employee e " +
                  "left join fetch e.projects " +
                  "where " +
                  "       e.username = :username and " +
                  "       e.password = :password",
            Employee.class)
      .setParameter("username", username)
      .setParameter("password", password)
      .getSingleResult();
```

Dynamic fetching via queries – using Criteria API

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();
CriteriaQuery<Employee> query = builder.createQuery( Employee.class );
Root<Employee> root = query.from( Employee.class );
root.fetch( "projects", JoinType.LEFT);
query.select(root).where(
        builder.and(
                builder.equal(root.get("username"), username),
                builder.equal(root.get("password"), password)
        )
);
Employee employee = entityManager.createQuery( query ).getSingleResult();
```

Dynamic fetching via JPA entity graph

```
@Entity(name = "Employee")
@NamedEntityGraph(name = "employee.projects",
        attributeNodes = @NamedAttributeNode("projects")
)


Employee employee = entityManager.find(
        Employee.class,
        userId,
        Collections.singletonMap(
                "javax.persistence.fetchgraph",
                entityManager.getEntityGraph( "employee.projects" )
        )
);}
```

Dynamic fetching via JPA entity graph

- When executing a JPQL query, if an EAGER association is omitted, Hibernate will issue a secondary select for every association needed to be fetched eagerly, which can lead to N+1 query issues.

- For this reason, it's better to use LAZY associations, and only fetch them eagerly on a per-query basis.

Dynamic fetching via JPA entity graph – using @NamedSubgraph

```
@Entity(name = "Project")
@NamedEntityGraph(name = "project.employees",
        attributeNodes = @NamedAttributeNode(
                value = "employees",
                subgraph = "project.employees.department"
        ),
        subgraphs = @NamedSubgraph(
                name = "project.employees.department",
                attributeNodes = @NamedAttributeNode( "department" )
        )
)
public static class Project {

    @Id
    private Long id;

    @ManyToMany
    private List<Employee> employees = new ArrayList<>();
}
```

Dynamic fetching via JPA entity graph – using @NamedSubgraph

```
Project project = doInJPA( this::entityManagerFactory,

entityManager -> {
    return entityManager.find(
            Project.class,
            1L,
            Collections.singletonMap(
                    "javax.persistence.fetchgraph",
                    entityManager.getEntityGraph( "project.employees" )
            )
    );
} );
```

```
select
p.id as id1_2_0_, e.id as id1_1_1_, d.id as id1_0_2_,
       e.accessLevel as accessLe2_1_1_,
e.department_id as departme5_1_1_,
       decrypt( 'AES', '00', e.pswd  ) as pswd3_1_1_,
e.username as username4_1_1_,
       p_e.projects_id as projects1_3_0__,
p_e.employees_id as employee2_3_0__
       from
Project p
inner join
Project_Employee p_e
on p.id=p_e.projects_id
inner join
Employee e
on p_e.employees_id=e.id
inner join
Department d
on e.department_id=d.id
where
p.id = ?

-- binding parameter [1] as [BIGINT] - [1]
```

Creating and applying JPA graphs from text representations

```
final EntityGraph<Project> graph = GraphParser.parse(
        Project.class,
        "employees( department )",
        entityManager
);

final EntityGraph<Movie> graph = GraphParser.parse(
        Movie.class,
        "cast.key( name )",
        entityManager
);

final EntityGraph<Ticket> graph = GraphParser.parse(
        Ticket.class,
        "showing.key( movie( cast ) )",
        entityManager
);
```

Subtypes and Subgraphs. Combining Subgraphs

```
Class<Invoice> invoiceClass = ...;
javax.persistence.EntityGraph<Invoice> invoiceGraph = entityManager.createEntityGraph( invoiceClass );
invoiceGraph.addAttributeNode( "responsibleParty" );
invoiceGraph.addSubgraph( "responsibleParty" ).addAttributeNode( "taxIdNumber" );
invoiceGraph.addSubgraph( "responsibleParty", Corporation.class ).addAttributeNode( "ceo" );
invoiceGraph.addSubgraph( "responsibleParty", NonProfit.class ).addAttributeNode( "sector" );

final EntityGraph<Project> a = GraphParser.parse(
        Project.class, "employees( username )", entityManager
);
final EntityGraph<Project> b = GraphParser.parse(
        Project.class, "employees( password, accessLevel )", entityManager
);
final EntityGraph<Project> c = GraphParser.parse(
        Project.class, "employees( department( employees( username ) ) )", entityManager
);

final EntityGraph<Project> all = EntityGraphs.merge( entityManager, Project.class, a, b, c );
```

Dynamic fetching via Hibernate profiles

```
@Entity(name = "Employee")
	@FetchProfile(
		name = "employee.projects",
		fetchOverrides = {
			@FetchProfile.FetchOverride(
				entity = Employee.class,
				association = "projects",
				mode = FetchMode.JOIN
			)
		}
	)
```

```
session.enableFetchProfile( "employee.projects" );
Employee employee = session.bySimpleNaturalId( Employee.class ).load( username );
```

Batch fetching - I

```
@Entity(name = "Department")
public static class Department {
    @Id
    private Long id;
    @OneToMany(mappedBy = "department")
    //@BatchSize(size = 5)
    private List<Employee> employees = new ArrayList<>();
}

@Entity(name = "Employee")
public static class Employee {
    @Id
    private Long id;
    @NaturalId
    private String name;
    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;
}
```

- Without @BatchSize, you'd run into a N+1 query issue, so, instead of 2 SQL statements, there would be 10 queries needed for fetching the Employee child entities.

- However, although @BatchSize is better than running into an N+1 query issue, most of the time, a DTO projection or a JOIN FETCH is a much better alternative since it allows you to fetch all the required data with a single query.

Batch fetching - II

```
List<Department> departments = entityManager.createQuery(
                "select d " +
                        "from Department d " +
                        "inner join d.employees e " +
                        "where e.name like 'John%'", Department.class)
        .getResultList();
for ( Department department : departments ) {
    log.infof(
            "Department %d has {} employees",
            department.getId(),
            department.getEmployees().size()
    );
}
SELECT
d.id as id1_0_
        FROM
Department d
INNER JOIN
Employee employees1_
ON d.id=employees1_.department_id
```

```
 SELECT
e.department_id as departme3_1_1_,
        e.id as id1_1_1_,
e.id as id1_1_0_,
        e.department_id as departme3_1_0_,
e.name as name2_1_0_
        FROM
Employee e
WHERE e.department_id IN ( 0, 2, 3, 4, 5)
SELECT
e.department_id as departme3_1_1_,
        e.id as id1_1_1_,
e.id as id1_1_0_,
        e.department_id as departme3_1_0_,
e.name as name2_1_0_
        FROM
Employee e
WHERE  e.department_id IN ( 6, 7, 8, 9, 1)
```

The @Fetch annotation mapping – FetchMode s:

- SELECT - the association is going to be fetched lazily using a secondary select for each individual entity, collection, or join load. It's equivalent to JPA FetchType.LAZY fetching strategy.
- JOIN - use an outer join to load the related entities, collections or joins when using direct fetching. It's equivalent to JPA FetchType.EAGER fetching strategy.
- SUBSELECT - available for collections only. When accessing a non-initialized collection, this fetch mode will trigger loading all elements of all collections of the same role for all owners associated with the persistence context using a single secondary select.

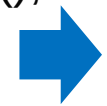
Dynamic fetching via Hibernate profiles

```
@OneToMany(mappedBy = "department", fetch = FetchType.LAZY)
    @Fetch(FetchMode.SUBSELECT)
    private List<Employee> employees = new ArrayList<>();

    List<Department> departments = entityManager.createQuery(
                "select d " +
                        "from Department d " +
                        "where d.name like :token", Department.class )
            .setParameter( "token", "Department%" )
            .getResultList();

    log.infof( "Fetched %d Departments", departments.size());

    for (Department department : departments ) {
        assertEquals( 3, department.getEmployees().size() );
    }
```

```
SELECT
    d.id as id1_0_
        FROM
    Department d
    where
    d.name like 'Department%'
        -- Fetched 2 Departments

        SELECT
    e.department_id as departme3_1_1_,
        e.id as id1_1_1_,
    e.id as id1_1_0_,
        e.department_id as departme3_1_0_,
    e.username as username2_1_0_
        FROM
    Employee e
    WHERE
    e.department_id in (
        SELECT
    fetchmodes0_.id
        FROM
    Department fetchmodes0_
    WHERE
    d.name like 'Department%'
)
```

@LazyCollection

```
@Entity(name = "Department")
public static class Department {
    @Id  private Long id;

    @OneToMany(mappedBy = "department",
            cascade = CascadeType.ALL)
    @OrderColumn(name = "order_id")
    @LazyCollection( LazyCollectionOption.EXTRA )
    private List<Employee> employees = new ArrayList<>();
}

@Entity(name = "Employee")
public static class Employee {
    @Id    private Long id;

    @NaturalId
    private String username;
    @ManyToOne(fetch = FetchType.LAZY)
    private Department department;
}
```



- TRUE - Load it when the state is requested.
- FALSE - Eagerly load it.
- EXTRA - Prefer extra queries over full collection loading.
- LazyCollectionOption.EXTRA only works for ordered collections, either List(s) that are annotated with @OrderColumn or Map(s).
- For bags (e.g. regular List(s) of entities that do not preserve any certain ordering), the @LazyCollection(LazyCollectionOption.EXTRA) behaves like any other FetchType.LAZY collection (the collection is fetched entirely upon being accessed for the first time).

Criteria API and Metamodel API

- Criteria queries offer a type-safe alternative to HQL, JPQL and native SQL queries.
- JPA 2 defines a typesafe Criteria API which allows Criteria queries to be constructed in a strongly-typed manner, utilizing so called static metamodel classes. For developers it is important that the task of the metamodel generation can be automated.
- Hibernate Static Metamodel Generator is an annotation processor based on JSR_269 with the task of creating JPA 2 static metamodel classes. The following example shows two JPA 2 entities Order and Item, together with the metamodel class Order_ and a typesafe query.

@LazyCollection

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<Person> criteria = builder.createQuery( Person.class );
Root<Person> root = criteria.from( Person.class );
criteria.select( root );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<Person> persons = entityManager.createQuery( criteria ).getResultList();
```

```
CriteriaBuilder builder = entityManager.getCriteriaBuilder();

CriteriaQuery<String> criteria = builder.createQuery( String.class );
Root<Person> root = criteria.from( Person.class );
criteria.select( root.get( Person_.nickName ) );
criteria.where( builder.equal( root.get( Person_.name ), "John Doe" ) );

List<String> nickNames = entityManager.createQuery( criteria ).getResultList();
```

References

- [PoEAA] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Professional. 2002.
- [JPwH] Christian Bauer & Gavin King. Java Persistence with Hibernate, Second Edition. Manning. 2015.

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD